DYNAMIC ENGINEERING
150 DuBois St., Suite C Santa Cruz, CA 95060
(831) 457-8891 **Fax** (831) 457-4793
http://www.dyneng.com
sales@dyneng.com
Est. 1988

PMC-Biserial-III NG8 Base & Channel

Driver Documentation

Win32 Driver Model

Manual Revision A
Corresponding Hardware: Revision D
10-2005-0204
Corresponding Firmware:
NG8: Design B, Revision 1

NG8Base & NG8Chan
WDM Device Drivers for the
PMC-Biserial-III-NG8

Dynamic Engineering
150 DuBois St., Suite C
Santa Cruz, CA 95060
(831) 457-8891
FAX: (831) 457-4793

Manual Revision A Revised Nov 23, 2009

Table of Contents

Introduction

The NG8Base and NG8Chan drivers are Win32 driver model (WDM) device drivers for the PmcBis3Ng8 from Dynamic Engineering.

The NG8 driver package has two parts. The driver is installed into the Windows® OS, and the User Application "Userap" executable.

The driver is delivered as installed or executable items to be used directly or indirectly by the user. The Userap code is delivered in source form [C] and is for the purpose of providing a reference to using the driver.

UserAp is a stand-alone code set with a simple, and powerful menu plus a series of "tests" that can be run on the installed hardware. Each of the tests execute calls to the driver, pass parameters and structures, and get results back. With the sequence of calls demonstrated, the functions of the hardware are utilized for loop-back testing. The software is used for manufacturing test at Dynamic Engineering. For example most Dynamic Engineering PCI based designs support DMA. DMA is demonstrated with the memory based loop-back tests. The tests can be ported and modified to fit your requirements.

The test software can be ported to your application to provide a running start. It is recommended to port the switch and status tests to your application to get started. The tests are simple and will quickly demonstrate the end-to-end operation of your application making calls to the driver and interacting with the hardware.

The menu allows the user to add tests, to run sequences of tests, to run until a failure occurs and stop or to continue, to program a set number of loops to execute and more. The user can add tests to the provided test suite to try out application ideas before committing to your system configuration. In many cases the test configuration will allow faster debugging in a more controlled environment before integrating with the rest of the system.

The hardware has features common to the board level and features that are set apart in "channels". The channels have the same offsets within the channel, and the same status and control bit locations allowing for symmetrical software in the calling routines. The driver supports the channels with a variable passed in to identify which channel is being accessed. The hardware manual defines the pinout for each channel and the bitmaps and detailed configurations for each channel. The driver handles all aspects of interacting with the channels and base features.

We strive to make a useable product, and while we can guarantee operation we can't foresee all concepts for client implementation. If you have suggestions for extended features, special calls for particular set-ups or whatever please share them with us,

[engineering@dyneng.com] and we will consider and in many cases add them.

The PmcBis3Ng8 design has a Spartan3 Xilinx FPGA to implement the PCI interface, FIFO's and protocol control and status for the IO. The IO are grouped into two channel's. Transmission and Reception can be accomplished with either or both channels under software control. Please refer to the HW manual for a much more complete description of the HW features.

When the PmcBis3Ng8 board is recognized by the PCI bus configuration utility it will start the NG8Base driver which will create a device object for each board, initialize the hardware, create a child devices for the channel(s) and request loading of the NG8Chan driver. The NG8Chan driver will create a device object for the I/O channel(s) and perform initialization on the channel(s). IO Control calls (IOCTLs) are used to configure the board and read status. Read and Write calls are used to move blocks of data in and out of the device.

Note:

This documentation will provide information about all calls made to the drivers, and how the drivers interact with the device for each of these calls. For more detailed information on the hardware implementation, refer to the PmcBis3Ng8 user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. These files include driver: NG8Base.sys, PmcBis3NG8.inf, DDNG8Base.h, NG8BaseGUID.h, NG8Chan.sys, DDNG8Chan.h, NG8ChanGUID.h. Userap: User Application source files.

NG8BaseGUID.h and NG8ChanGUID.h are C header files that define the device interface identifiers for the drivers. DDNG8Base.h and DDNG8Chan.h files are C header files that define the Application Program Interface (API) to the drivers. These files are required at compile time by any application that wishes to interface with the drivers, but they are not needed for driver installation. The files are included with the Userap fileset.

Windows 2000 Installation

Copy PmcBis3NG8.inf, NG8Base.sys and NG8Chan.sys to a floppy disk, or CD if preferred. In some cases the files can be accessed over a network or from local HDD. Substitute the network address for the floppy instructions to proceed with an over the network installation. The files are stored at the root of the PmcBis3Ng8Userap file set.

With the hardware installed, power-on the PCI host computer and wait for the ***Found New Hardware Wizard*** dialogue window to appear.
_ Select ***Next***.
_ Select ***Search for a suitable driver for my device.***
_ Select ***Next***.
_ Insert the disk prepared above in the desired drive.
_ Select the appropriate drive e.g. ***Floppy disk drives***.
_ Select ***Next***.
_ The wizard should find the PmcBis3NG8.inf file.
_ Select ***Next***.
_ Select ***Finish*** to close the ***Found New Hardware Wizard***.
The system should now see the channels and reopen the ***New Hardware Wizard.***
Repeat this for each channel as necessary.

Windows XP Installation

Copy PmcBis3NG8.inf, NG8Base.sys and NG8Chan.sys to a floppy disk, or CD if preferred. In some cases the files can be accessed over a network or from local HDD. Substitute the network address for the floppy instructions to proceed with an over the network installation.

With the hardware installed, power-on the PCI host computer and wait for the ***Found New Hardware Wizard*** dialogue window to appear.
_ Insert the disk prepared above in the desired drive.
_ Select ***No when asked to connect to Windows Update***.
_ Select ***Next***.
_ Select ***Install the software automatically.***
_ Select ***Next***.
_ Select ***Finish*** to close the ***Found New Hardware Wizard***.
The system should now see the channels and reopen the ***New Hardware Wizard.***
Proceed as above for each channel as necessary.

Driver Startup

Once the drivers have been installed they will start automatically when the system recognizes the hardware.

Handles can be opened to a specific board by using the CreateFile() function call and passing in the device names obtained from the system.

The interfaces to the devices are identified using globally unique identifiers (GUIDs), which are defined in NG8BaseGUID.h and NG8ChanGUID.h.

The User Application software contains a file called “main.c”. Main has the initialization needed to get the handles to the base and channel assets of the installed PmcBis3Ng8 device.

The main file provided is designed to work with our test menu and includes user interaction steps to allow the user to select which board is being tested in a multiple board environment. The integrator can hardcode for single board systems or use an automatic loop to operate in multiple board systems without using user interaction. For multiple user systems it is suggested that the board number is associated with a switch setting so the calls can be associated with a particular board from a physical point of view. The DIPswitch is provided for this purpose.

IO Controls

The drivers use IO Control calls (IOCTLs) to configure the device. IOCTLs refer to a single Device Object, which controls a single board or I/O channel. IOCTLs are called using the Win32 function DeviceIoControl() (see below), and passing in the handle to the device opened with CreateFile() (see above). IOCTLs generally have input parameters, output parameters, or both. Often a custom structure is used.

```
BOOL DeviceIoControl(
  HANDLE        hDevice,          // Handle opened with
CreateFile()
  DWORD         dwIoControlCode, // Control code defined in API
header file
  LPVOID        lpInBuffer,       // Pointer to input parameter
  DWORD         nInBufferSize,    // Size of input parameter
  LPVOID        lpOutBuffer,      // Pointer to output parameter
  DWORD         nOutBufferSize,   // Size of output parameter
  LPDWORD       lpBytesReturned, // Pointer to return length
parameter
  LPOVERLAPPED lpOverlapped,     // Optional pointer to
overlapped structure
);                                 // used for asynchronous I/O
```

The IOCTLs defined for the NG8Base driver are described below:
Please note that the address map is included in the DD file for reference when writing your own driver for a different OS.

IOCTL_NG8_BASE_GET_INFO
Function: Return the Instance Number, Switch value, PLL device ID, Xilinx rev and Current Driver Version
Input: None
Output: NG8_BASE_DRIVER_DEVICE_INFO : Structure
Notes: Switch value is the configuration of the on-board dip-switch that has been set by the User (see the board silk screen for bit position and polarity). The PLL ID is the device address of the PLL device. This value, which is set at the factory, is usually 0x69 but may also be 0x6A. See DDNG8Base.h for the definition of NG8_BASE_DRIVER_DEVICE_INFO.

IOCTL_NG8_BASE_LOAD_PLL_DATA
Function: Loads the internal registers of the PLL.
Input: NG8_BASE_PLL_DATA structure
Output: None
Notes:

IOCTL_NG8_BASE_READ_PLL_DATA
Function: Returns the contents of the PLL's internal registers
Input: None
Output: NG8_BASE_PLL_DATA structure
Notes: The register data is output in the NG8_BASE_PLL_DATA structure in an array of 40 bytes.

IOCTL_NG8_BASE_SET_BASEREG
Function: Write to Base Control Register - general access to base control register of card, use with bit definitions
Input: ULONG
Output: none
Notes: Use for general purpose – bit mapped access to the base control register.

IOCTL_NG8_BASE_GET_BASEREG
Function: Read from Base Control Register - general access from base control register of card, use with bit definitions
Input: none
Output: ULONG
Notes: Use for general purpose – bit mapped access to the base control register.

IOCTL_NG8_BASE_GET_STATUS
Function: Read from Status Register
Input: none
Output: ULONG
Notes: Use for general purpose – bit mapped access from the register. See DDNG8Base.h for bit map information. See the HW manual for exact definitions of bits.

The IOCTLs defined for the NG8Chan driver are described below:
In the NG8 implementation both the Transmitter and the Receiver interface are implemented within the same channel (0,1). The Receiver accepts data from the external equipment. The Transmitter provides data to the external equipment. Loop-back can be accomplished across channels 0=>1 or 1=>0.

Address and bit map information is included in the DDNG8Chan.h file to support those who are writing drivers for other OS.

IOCTL_NG8_CHAN_GET_INFO
Function: Return the Instance Number and Current Driver Version
Input: None
Output: NG8_CHAN_DRIVER_DEVICE_INFO structure
Notes: See the definition of NG8_CHAN_DRIVER_DEVICE_INFO in the DDNG8Chan.h header file.

IOCTL_NG8_CHAN_GET_STATUS
Function: Return the value of the status register and clear latched bits
Input: None
Output: Status register value(ULONG)
Notes: Latched interrupt and error status are cleared by write-back. See quick reference status bits below. Defines available in DDNG8Chan.h Detailed definitions are available in the HW manual.

#define STAT_TX_FIFO_MT	0x00000001 //0 set when TX DMA FIFO MT
#define STAT_TX_FIFO_AE	0x00000002 //1 set when TX DMA FIFO AMT
#define STAT_TX_FIFO_FULL	0x00000004 //2 set when TX DMA FIFO is Full
#define STAT_RX_FIFO_MT	0x00000010 //4 set when RX FIFO is MT
#define STAT_RX_FIFO_AF	0x00000020 //5 set when RX FIFO is AFull
#define STAT_RX_FIFO_FULL	0x00000040 //6 set when RX FIFO is Full
#define STAT_TX_AMT_INT	0x00000100 //8 Set when total data count in TX path is less than programmed level
#define STAT_RX_AFL_INT	0x00000200 //9 Set when total data count in RX path is more than programmed level
#define STAT_TX_AE_INT_LAT	0x00000400 //10 Transmit FIFO Interrupt Occurred, latched - clear with write
#define STAT_RX_AF_INT_LAT	0x00000800 //11 Receive FIFO Interrupt Occurred, latched - clear with write
#define STAT_WR_DMA_ERR	0x00001000 //12 write DMA error, latched - clear with write

```
#define STAT_RD_DMA_ERR          0x00002000 //13 read DMA error, latched - clear with write
#define STAT_WR_DMA_INT          0x00004000 //14 write DMA Interrupt, latched - clear with write
#define STAT_RD_DMA_INT          0x00008000 //15 read DMA Interrupt, latched - clear with write

#define STAT_TX_IMAGE_COMPLETE   0x00010000 //16 set when TX image completed, latched - clear with write, set once per image sent
#define STAT_RX_IMAGE_COMPLETE   0x00020000 //17 set when RX image completed, latched - clear with write, set once per image received
#define STAT_RX_OVFL_ERR         0x00040000 //18 Set when RX overflow error occurred, latched - clear with write
#define STAT_TX_UNFL_ERR         0x00080000 //19 Set when TX underflow error occurred, latched - clear with write

#define STAT_RX_IDLE             0x00100000 //20 set when RX is in Idle state
#define STAT_TX_IDLE             0x00200000 //21 set when TX is in Idle state
#define STAT_DMA_RD_IDLE         0x00400000 //22 set when Burst Out [read] DMA state-machine is in the idle state
#define STAT_DMA_WR_IDLE         0x00800000 //23 set when Burst In [write] DMA state-machine is in the idle state

#define STAT_EXT_FIFO_MT         0x01000000 //24 Ext FIFO is MT when set
#define STAT_EXT_FIFO_AMT        0x02000000 //25 Ext FIFO is AMT when set
#define STAT_EXT_FIFO_AFULL      0x04000000 //26 Ext FIFO is AFL when set
#define STAT_EXT_FIFO_FULL       0x08000000 //27 Ext FIFO is FULL when set

#define STAT_DIRECTION           0x10000000 //28 Set for transmit, cleared for receive => copy of control bit for this design
#define LOCAL_INT                0x40000000 //30 non DMA interrupt status before channel mask
#define STAT_ACTIVE_INT          0x80000000 //31 channel interrupt is active [after mask and includes DMA]
```

IOCTL_NG8_CHAN_CLR_STATUS
Function: Clear Error Bits latched and not cleared by status read
Input: ULONG
Output: none
Notes: Clear latched error bits. Allows polling on FIFO status without losing potential Error conditions. Write back with same bit position set to clear. Defines available in DDNG8Chan.h Detailed definitions are available in the HW manual.

IOCTL_NG8_CHAN_SET_FIFO_LEVELS
Function: Sets the transmitter almost empty and receiver almost full levels for the channel.
Input: NG8_CHAN_FIFO_LEVELS structure
Output: None
Notes: The FIFO counts are compared to these levels to determine the value of the STAT_TX_FF_AE and STAT_RX_FF_AF status bits.

IOCTL_NG8_CHAN_GET_FIFO_LEVELS
Function: Returns the transmitter almost empty and receiver almost full levels for the channel.
Input: None
Output: NG8_CHAN_FIFO_LEVELS structure
Notes:

IOCTL_NG8_CHAN_GET_FIFO_COUNTS
Function: Returns the number of data words in FIFO's.
Input: None
Output: NG8_CHAN_FIFO_COUNTS structure
Notes: Returns the actual TX FIFO data counts and count including DMA pipeline RX FIFO, plus external and state machine FIFO levels.

IOCTL_NG8_CHAN_RESET_FIFOS
Function: Resets one or all FIFO's for the referenced channel.
Input: NG8_FIFO_SEL enumeration type See structure definition in DDNG8Chan.h
Output: None
Notes: Resets Transmit, Receive, External or All FIFO's. Please note the Tx / Rx state machines are also reset by this command.

IOCTL_NG8_CHAN_REGISTER_EVENT

Function: Registers an event to be signaled when an interrupt occurs.
Input: Handle to the Event object
Output: None
Notes: The caller creates an event with CreateEvent() and supplies the handle returned from that call as the input to this IOCTL. The driver then obtains a system pointer to the event and signals the event when a user interrupt is serviced. The user interrupt service routine waits on this event, allowing it to respond to the interrupt. The DMA interrupts do not cause the event to be signaled.

IOCTL_NG8_CHAN_ENABLE_INTERRUPT

Function: Enables the channel Master Interrupt.
Input: None
Output: None
Notes: This command must be run to allow the board to respond to user interrupts. The master interrupt enable is disabled in the driver interrupt service routine when a user interrupt is serviced. Therefore this command must be run after each interrupt occurs to re-enable it.

IOCTL_NG8_CHAN_DISABLE_INTERRUPT

Function: Disables the channel Master Interrupt.
Input: None
Output: None
Notes: This call is used when user interrupt processing is no longer desired.

IOCTL_NG8_CHAN_FORCE_INTERRUPT

Function: Causes a system interrupt to occur.
Input: None
Output: None
Notes: Causes an interrupt to be asserted on the PCI bus as long as the channel master interrupt is enabled. This IOCTL is used for development, to test interrupt processing. Board level master interrupt also needs to be set.

IOCTL_NG8_CHAN_GET_ISR_STATUS

Function: Returns the interrupt status read in the ISR from the last user interrupt.
Input: None
Output: Interrupt status value (unsigned long integer)
Notes: Returns the interrupt status that was read in the interrupt service routine of the last interrupt caused by one of the enabled channel interrupts. The interrupts that deal with the DMA transfers do not affect this value. Masked version of channel status.

IOCTL_NG8_CHAN_SWW_TX_FIFO
Function: Writes a 32-bit data word to the transmit FIFO.
Input: FIFO word (unsigned long integer)
Output: none
Notes: Used to make single-word accesses to the transmit FIFO instead of using DMA.

IOCTL_NG8_CHAN_SWR_RX_FIFO
Function: Returns a 32-bit data word from the receive FIFO.
Input: None
Output: FIFO word (unsigned long integer)
Notes: Used to make single-word accesses to the receive FIFO instead of using DMA. Please note, Data read from this port is no longer available in the FIFO for DMA or other use.

IOCTL_NG8_CHAN_SET_CONTROL
Function: write to Channel Control register using structure
Input: NG8_CHAN_CONT
Output: None
Notes: See DDNG8Chan.h for structure. See below for quick reference.

IOCTL_NG8_CHAN_GET_CONTROL
Function: Read from Channel Control register using structure
Input: None
Output: NG8_CHAN_CONT
Notes: See DDNG8Chan.h for structure. See below for quick reference.
FifoTestEn; // BiPass Mode Control
MIntEn; // Master Interrupt Enable
WrDmaEn; // Write DMA Interrupt Enable
RdDmaEn; // Read DMA Interrupt Enable
TxUrgent; // Set for higher priority TX DMA processing
RxUrgent; // Set for higher priority RX DMA processing
ExtFifoLoad;// control access to Ext FIFO programmable level registers or data array
ExtFifoMux; // 0 = Path A(TX), 1 = Path B(RX) at input mux to External FIFO
Direction; // 0 = RX mode, 1 = TX mode. Controls

IOCTL_NG8_CHAN_SET_TX
Function: write to Channel Tx Control register using structure
Input: NG8_CHAN_TX_CONTROL
Output: None
Notes: See DDNG8Chan.h for structure.

IOCTL_NG8_CHAN_GET_TX
Function: Read from Channel Master Control register using structure
Input: None
Output: NG8_CHAN_TX_CONTROL
Notes: See DDNG8Chan.h for structure.

Quick Reference:

TxStart;	//0 start TX state machine
TxFifoEn;	//1 set to begin enable data transfer from ext fifo to TX fifo
TxIntEn;	//2 set to enable TX interrupt - occurs when each image is transmitted
TxAEIntEn;	//3 set to enable TX FIFO based interrupt [almost empty] based on latched status (requires not almost empty to become almost mt
TxUnFlEn;	//4 set to enable UnderFlow interrupt
TxByteOrder;	//5 set to use upper [D31-16] then lower[D15-0], clear to use lower then upper
TxAEIntEnLvl;	//6 set to enable Almost Empty FIFO interrupt based on AE level. Interrupt active whenever almost empty is true

IOCTL_NG8_CHAN_SET_TX_COUNT
Function: write to Channel TXCount register
Input: ULONG
Output: None
Notes: Set the count for the Transmit Image size in Pixels/2. There are 2 pixels per word loaded.

IOCTL_NG8_CHAN_GET_TX_COUNT
Function: Read from Channel TX Count Register
Input: None
Output: ULONG
Notes:

IOCTL_NG8_CHAN_SET_TX_READY
Function: write to Channel TX Ready Count Register
Input: ULONG
Output: None
Notes: Set the count for starting the Transmitter. When the transmitter is enabled the State-machine waits for the READY COUNT level matched against the total FIFO path for TX and for the SM FIFO almost full conditions to make sure the pipeline has enough data to begin transmission without an UnderFlow error.

IOCTL_NG8_CHAN_GET_TX_READY
Function: Read from Channel TX Ready Count Register
Input: None
Output: ULONG
Notes: Read back port.

IOCTL_NG8_CHAN_SET_TX_AMT
Function: write to Channel TX Almost Empty Count Register
Input: ULONG
Output: None
Notes: Set the count for comparing against the TX FIFO path total count for the Almost Empty condition. The Almost Empty level interrupt control count is set with this register.

IOCTL_NG8_CHAN_GET_TX_AMT
Function: Read from Channel TX Almost Empty Count Register
Input: None
Output: ULONG
Notes: Read back port.

IOCTL_NG8_CHAN_TX_FIFO_WORDCNT_READ
Function: Read Level of Data stored in Transmit Local FIFO
Input: None
Output: ULONG
Notes: Read only. 1Kx32 max value.

IOCTL_NG8_CHAN_EXT_FIFO_WORDCNT_READ
Function: Read Level of Data stored in External FIFO
Input: None
Output: ULONG
Notes: Read only. 128Kx32 max value. Please note the External FIFO is used for both RX and TX functions.

IOCTL_NG8_CHAN_TX_FIFO_TOTALWORDCNT_READ
Function: Read Combined Level of Data stored in DMA, External and TX SM FIFO's
Input: None
Output: ULONG
Notes: Read only. 133Kx32 max value. Please note that values in the External FIFO will show in this count even if channel is set to RX mode.

IOCTL_NG8_CHAN_SET_RX
Function: write to Channel Receiver Control register using structure
Input: NG8_CHAN_RX_CONTROL
Output: None
Notes: See DDNG8Chan.h for structure.

IOCTL_NG8_CHAN_GET_RX
Function: Read from Channel Receiver Control register using structure
Input: None
Output: NG8_CHAN_RX_CONTROL
Notes: See DDNG8Chan.h for structure.

Quick Reference:

RxStart;	//0 set to begin RX Data Acquisition
RxIntEn;	//2 set to enable RX interrupt for each image captured
RxAFIntEn;	//3 set to enable RX FIFO based interrupt [almost full]
RxOvFlEn;	//4 set to enable RX OverFlow interrupt
RxByteOrder;	//5 set to reverse bytes after receiving
RxAFIntEnLvl;	//6 set to enable Almost Full FIFO interrupt based on AF level. Interrupt active whenever almost Full is true
RxFifoPathEn;	//10 set to enable data movement from RX FIFO to External FIFO and from External FIFO to RX DMA FIFO

IOCTL_NG8_CHAN_SET_RX_COUNT
Function: write to Channel Receiver Count register
Input: ULONG
Output: None
Notes: Set the count for the size of the image to be received. The COUNT is the number of LW written to the FIFO per image; Pixels/2. The receiver will wait for R0C0 [both HREF and VREF asserted] and then receive data. When the COUNT has been loaded into the FIFO the Complete status is set. The hardware will look for the next image R0C0 synchronization. The hardware runs at a higher rate than the received data allowing back-to-back transfers.

IOCTL_NG8_CHAN_GET_RX_COUNT
Function: Read from Channel Receiver Count register
Input: None
Output: ULONG
Notes:

IOCTL_NG8_CHAN_SET_RX_AFL
Function: write to Channel Receiver Almost Full Register
Input: ULONG
Output: None
Notes: Set the level to compare against for the Almost Full Level interrupt. The level is compared against the total RX FIFO path data and when the FIFO path has more data than the AFL register value the AFL interrupt and status are set.

IOCTL_NG8_CHAN_GET_RX_AFL
Function: Read from Channel Receiver Almost Full Register
Input: None
Output: ULONG
Notes:

IOCTL_NG8_CHAN_RX_FIFO_TOTALWORDCNT_READ
Function: Read Combined Level of Data stored in DMA Pipeline, RX DMA, External and RX SM FIFO's
Input: None
Output: ULONG
Notes: Read only. 133Kx32 max value. Please note that values in the External FIFO will show in this count even if channel is set to TX mode. This count can be used to set "final" DMA read values when reception is stopped and the image count is not known.

Write

DMA data is written to the referenced I/O channel device using the write command. Writes are executed using the Win32 function WriteFile() and passing in the handle to the I/O channel device opened with CreateFile(), a pointer to a pre-allocated buffer containing the data to be written, an unsigned long integer that represents the size of that buffer in bytes, a pointer to an unsigned long integer to contain the number of bytes actually written, and a pointer to an optional Overlapped structure for performing asynchronous IO.

Read

DMA data is read from the referenced I/O channel device using the read command. Reads are executed using the Win32 function ReadFile() and passing in the handle to the I/O channel device opened with CreateFile(), a pointer to a pre-allocated buffer that will contain the data read, an unsigned long integer that represents the size of that buffer in bytes, a pointer to an unsigned long integer to contain the number of bytes actually read, and a pointer to an optional Overlapped structure for performing asynchronous IO.

Examples of using DMA are provided in the reference software FIFO and IO loop-tests.

Warranty and Repair

Dynamic Engineering warrants this product to be free from defects under normal use and service and in its original, unmodified condition, for a period of one year from the time of purchase. If the product is found to be defective within the terms of this warranty, Dynamic Engineering's sole responsibility shall be to repair, or at Dynamic Engineering's sole option to replace, the defective product.

Dynamic Engineering's warranty of and liability for defective products is limited to that set forth herein. Dynamic Engineering disclaims and excludes all other product warranties and product liability, expressed or implied, including but not limited to any implied warranties of merchandisability or fitness for a particular purpose or use, liability for negligence in manufacture or shipment of product, liability for injury to persons or property, or for any incidental or consequential damages.

Dynamic Engineering’s products are not authorized for use as critical components in life support devices or systems without the express written approval of the president of Dynamic Engineering.

Service Policy

Before returning a product for repair, verify as well as possible that the driver is at fault. The driver has gone through extensive testing and in most cases it will be “cockpit error” rather than an error with the driver. When you are sure or at least willing to pay to have someone help then call the Customer Service Department and arrange to speak with an engineer. We will work with you to determine the cause of the issue. If the issue is one of a defective driver we will correct the problem and provide an updated module(s) to you [no cost]. If the issue is of the customer’s making [anything that is not the driver] the engineering time will be invoiced to the customer. Pre-approval may be required in some cases depending on the customer’s invoicing policy.

Out of Warranty Repairs

Out of warranty support will be billed. The current minimum repair charge is $125. An open PO will be required.

For Service Contact:

Customer Service Department
Dynamic Engineering
150 DuBois Street, Suite C
Santa Cruz, CA 95060
831-457-8891
831-457-4793 Fax

support@dyneng.com

Appendix

Reference copy of structures for evaluation

The following structures shown are available in the DDNG8Chan.h and DDNG8Base.h files included with the driver. The structures are included here for your evaluation when considering the driver package. The electronic versions included with the driver should be used with your project. The names track the register bit definitions. For details about particular signals please refer to the HW manual.

Base:

```
#define PLL_MESSAGE1_SIZE       16
#define PLL_MESSAGE2_SIZE       24
#define PLL_MESSAGE_SIZE        (PLL_MESSAGE1_SIZE + PLL_MESSAGE2_SIZE)

 // Driver/Device information
typedef struct _NG8_BASE_DRIVER_DEVICE_INFO
{
   UCHAR    DriverVersion;
   UCHAR    XilinxVersion;
   UCHAR    XilinxDesign;
   UCHAR    PllDeviceId;
   UCHAR    SwitchValue;
   ULONG    InstanceNumber;
} NG8_BASE_DRIVER_DEVICE_INFO, *PNG8_BASE_DRIVER_DEVICE_INFO;

typedef struct _NG8_BASE_PLL_DATA
{
   UCHAR    Data[PLL_MESSAGE_SIZE];
} NG8_BASE_PLL_DATA, *PNG8_BASE_PLL_DATA;
```

Channel:

```
typedef struct _NG8_CHAN_DRIVER_DEVICE_INFO
{
  UCHAR    DriverVersion;
  ULONG    InstanceNumber;
} NG8_CHAN_DRIVER_DEVICE_INFO, *PNG8_CHAN_DRIVER_DEVICE_INFO;

typedef enum _NG8_CHAN_FIFO_SEL {NG8_RX, NG8_TX, NG8_EXT, NG8_ALL}
NG8_CHAN_FIFO_SEL, *PNG8_CHAN_FIFO_SEL;

typedef struct _NG8_CHAN_FIFO_LEVELS
{
  USHORT   AlmostFull;              // Set to control Master HW with Almost full definition
  USHORT   AlmostEmpty;             // set to control Target HW with Almost Empty definition,
Also controls Interrupt request
} NG8_CHAN_FIFO_LEVELS, *PNG8_CHAN_FIFO_LEVELS;

typedef struct _NG8_CHAN_FIFO_COUNTS
{
  USHORT            RxCountwPipe;       // RX DMA FIFO plus pipeline
  USHORT            TxCount;            // TX DMA FIFO
  USHORT            TSMCount;           // TX State-Machine / holding FIFO
  USHORT            RSMCount;           // RX State-Machine / holding FIFO
  ULONG             ExtCount;           // External FIFO Count
  ULONG             TxTotalCount;       // Sum of TX DMA, EXT and SM counts
  ULONG             RxTotalCount;       // Sum of TX DMA. EXT, SM and pipeline counts
} NG8_CHAN_FIFO_COUNTS, *PNG8_CHAN_FIFO_COUNTS;

typedef struct _NG8_CHAN_CONT
{
  BOOLEAN           FifoTestEn;         // BiPass Mode Control
  BOOLEAN           MIntEn;             // Master Interrupt Enable
  BOOLEAN           WrDmaEn;            // Write DMA Interrupt Enable
  BOOLEAN           RdDmaEn;            // Read DMA Interrupt Enable
  BOOLEAN           TxUrgent;           // Set for higher priority TX DMA processing
  BOOLEAN           RxUrgent;           // Set for higher priority RX DMA processing
  BOOLEAN           ExtFifoLoad;        // Set/Clear to control access to Ext FIFO
                                        programmable level registers or data array
  BOOLEAN           ExtFifoMux;         // 0 = Path A(TX), 1 = Path B(RX) at input mux to
                                        External    FIFO
  BOOLEAN           Direction;          // 0 = RX mode, 1 = TX mode. Controls
                                        transceivers and terminations.
} NG8_CHAN_CONT, *PNG8_CHAN_CONT;
```

```
typedef struct _NG8_CHAN_RX_CONTROL
{
  BOOLEAN          RxStart;          //0 set to begin RX Data Acquisition
  BOOLEAN          RxIntEn;          //2 set to enable RX interrupt for each image
                                       captured
  BOOLEAN          RxAFIntEn;        //3 set to enable RX DMA FIFO based interrupt
                                       [almost full]
  BOOLEAN          RxOvFlEn;         //4 set to enable RX OverFlow interrupt
  BOOLEAN          RxByteOrder;      //5 set to reverse bytes after receiving
  BOOLEAN          RxAFIntEnLvl;     //6 set to enable Almost Full FIFO interrupt based
                                       on AF level.  Int active whenever AFull is true
  BOOLEAN          RxFifoPathEn;     //10 set to enable data movement from RX FIFO to
                                       Ext FIFO & from Ext FIFO to RX DMA FIFO
} NG8_CHAN_RX_CONTROL, *PNG8_CHAN_RX_CONTROL;


typedef struct _NG8_CHAN_TX_CONTROL
{
  BOOLEAN          TxStart;          //0 start TX state machine
  BOOLEAN          TxFifoEn;         //1 set to begin enable data transfer from ext fifo to
                                       TX fifo
  BOOLEAN          TxIntEn;          //2 set to enable TX interrupt - occurs when each
                                       image is transmitted
  BOOLEAN          TxAEIntEn;        //3 set to enable TX FIFO based interrupt [almost
                                       empty] based on latched status (requires not
                                       almost empty to become almost mt
  BOOLEAN          TxUnFlEn;         //4 set to enable UnderFlow interrupt
  BOOLEAN          TxByteOrder;      //5 set to use upper [D31-16] then lower[D15-0],
                                       clear to use lower then upper
  BOOLEAN          TxAEIntEnLvl;     //6 set to enable Almost Empty FIFO interrupt
                                       based on AE level.  Interrupt active whenever
                                       almost empty is true
} NG8_CHAN_TX_CONTROL, *PNG8_CHAN_TX_CONTROL;
```